**从一个简单的 JPA 示例开始**

本文主要讲述 Spring Data JPA，但是为了不至于给 JPA 和 Spring 的初学者造成较大的学习曲线，我们首先从 JPA 开始，简单介绍一个 JPA 示例；接着重构该示例，并引入 Spring 框架，这两部分不会涉及过多的篇幅，如果希望能够深入学习 Spring 和 JPA，可以根据本文最后提供的参考资料进一步学习。

自 JPA 伴随 Java EE 5 发布以来，受到了各大厂商及开源社区的追捧，各种商用的和开源的 JPA 框架如雨后春笋般出现，为开发者提供了丰富的选择。它一改之前 EJB 2.x 中实体 Bean 笨重且难以使用的形象，充分吸收了在开源社区已经相对成熟的 ORM 思想。另外，它并不依赖于 EJB 容器，可以作为一个独立的持久层技术而存在。目前比较成熟的 JPA 框架主要包括 Jboss 的 Hibernate EntityManager、Oracle 捐献给 Eclipse 社区的 EclipseLink、Apache 的 OpenJPA 等。

本文的示例代码基于 Hibernate EntityManager 开发，但是读者几乎不用修改任何代码，便可以非常容易地切换到其他 JPA 框架，因为代码中使用到的都是 JPA 规范提供的接口 / 类，并没有使用到框架本身的私有特性。示例主要涉及七个文件，但是很清晰：业务层包含一个接口和一个实现；持久层包含一个接口、一个实现、一个实体类；另外加上一个 JPA 配置文件和一个测试类。相关类 / 接口代码如下：

**清单 1. 实体类 AccountInfo.java**

|  |
| --- |
| @Entity  @Table(name = "t\_accountinfo")  public class AccountInfo implements Serializable {  private Long accountId;  private Integer balance;  // 此处省略 getter 和 setter 方法。  } |

**清单 2. 业务层接口 UserService.java**

|  |
| --- |
| public interface UserService {  public AccountInfo createNewAccount(String user, String pwd, Integer init);  } |

**清单 3. 业务层的实现类 UserServiceImpl.java**

|  |
| --- |
| public class UserServiceImpl implements UserService {  private UserDao userDao = new UserDaoImpl();  public AccountInfo createNewAccount(String user, String pwd, Integer init){  // 封装域对象  AccountInfo accountInfo = new AccountInfo();  UserInfo userInfo = new UserInfo();  userInfo.setUsername(username);  userInfo.setPassword(password);  accountInfo.setBalance(initBalance);  accountInfo.setUserInfo(userInfo);  // 调用持久层，完成数据的保存  return userDao.save(accountInfo);  }  } |

**清单 4. 持久层接口**

|  |
| --- |
| public interface UserDao {  public AccountInfo save(AccountInfo accountInfo);  } |

**清单 5. 持久层的实现类**

|  |
| --- |
| public class UserDaoImpl implements UserDao {  public AccountInfo save(AccountInfo accountInfo) {  EntityManagerFactory emf =  Persistence.createEntityManagerFactory("SimplePU");  EntityManager em = emf.createEntityManager();  em.getTransaction().begin();  em.persist(accountInfo);  em.getTransaction().commit();  emf.close();  return accountInfo;  }  } |

**清单 6. JPA 标准配置文件 persistence.xml**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">  <persistence-unit name="SimplePU" transaction-type="RESOURCE\_LOCAL">  <provider>org.hibernate.ejb.HibernatePersistence</provider>  <class>footmark.springdata.jpa.domain.UserInfo</class>  <class>footmark.springdata.jpa.domain.AccountInfo</class>  <properties>  <property name="hibernate.connection.driver\_class"  value="com.mysql.jdbc.Driver"/>  <property name="hibernate.connection.url"  value="jdbc:mysql://10.40.74.197:3306/zhangjp"/>  <property name="hibernate.connection.username" value="root"/>  <property name="hibernate.connection.password" value="root"/>  <property name="hibernate.dialect"  value="org.hibernate.dialect.MySQL5Dialect"/>  <property name="hibernate.show\_sql" value="true"/>  <property name="hibernate.format\_sql" value="true"/>  <property name="hibernate.use\_sql\_comments" value="false"/>  <property name="hibernate.hbm2ddl.auto" value="update"/>  </properties>  </persistence-unit>  </persistence> |

**清单 7. 本文使用如下的 main 方法进行开发者测试**

|  |
| --- |
| public class SimpleSpringJpaDemo {  public static void main(String[] args) {  new UserServiceImpl().createNewAccount("ZhangJianPing", "123456", 1);  }  } |

[**回页首**](http://www.ibm.com/developerworks/cn/opensource/os-cn-spring-jpa/#ibm-pcon)

**简述 Spring 框架对 JPA 的支持**

接下来我们引入 Spring，以展示 Spring 框架对 JPA 的支持。业务层接口 UserService 保持不变，UserServiceImpl 中增加了三个注解，以让 Spring 完成依赖注入，因此不再需要使用 new 操作符创建 UserDaoImpl 对象了。同时我们还使用了 Spring 的声明式事务：

**清单 8. 配置为 Spring Bean 的业务层实现**

|  |
| --- |
| @Service("userService")  public class UserServiceImpl implements UserService {  @Autowired  private UserDao userDao;  @Transactional  public AccountInfo createNewAccount(  String name, String pwd, Integer init) { …… }  } |

对于持久层，UserDao 接口也不需要修改，只需修改 UserDaoImpl 实现，修改后的代码如下：

**清单 9. 配置为 Spring Bean 的持久层实现**

|  |
| --- |
| @Repository("userDao")  public class UserDaoImpl implements UserDao {  @PersistenceContext  private EntityManager em;  @Transactional  public Long save(AccountInfo accountInfo) {  em.persist(accountInfo);  return accountInfo.getAccountId();  }  } |

**清单 10. Spring 配置文件**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans...>  <context:component-scan base-package="footmark.springdata.jpa"/>  <tx:annotation-driven transaction-manager="transactionManager"/>  <bean id="transactionManager"  class="org.springframework.orm.jpa.JpaTransactionManager">  <property name="entityManagerFactory" ref="entityManagerFactory"/>  </bean>  <bean id="entityManagerFactory" class=  "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">  </bean>  </beans> |

**清单 11. 改造后的基于 Spring 的开发者测试代码**

|  |
| --- |
| public class SimpleSpringJpaDemo{  public static void main(String[] args){  ClassPathXmlApplicationContext ctx =  new ClassPathXmlApplicationContext("spring-demo-cfg.xml");  UserDao userDao = ctx.getBean("userDao", UserDao.class);  userDao.createNewAccount("ZhangJianPing", "123456", 1);  }  } |

通过对比重构前后的代码，可以发现 Spring 对 JPA 的简化已经非常出色了，我们可以大致总结一下 Spring 框架对 JPA 提供的支持主要体现在如下几个方面：

* 首先，它使得 JPA 配置变得更加灵活。JPA 规范要求，配置文件必须命名为 persistence.xml，并存在于类路径下的 META-INF 目录中。该文件通常包含了初始化 JPA 引擎所需的全部信息。Spring 提供的 LocalContainerEntityManagerFactoryBean 提供了非常灵活的配置，persistence.xml 中的信息都可以在此以属性注入的方式提供。
* 其次，Spring 实现了部分在 EJB 容器环境下才具有的功能，比如对 @PersistenceContext、@PersistenceUnit 的容器注入支持。
* 第三，也是最具意义的，Spring 将 EntityManager 的创建与销毁、事务管理等代码抽取出来，并由其统一管理，开发者不需要关心这些，如前面的代码所示，业务方法中只剩下操作领域对象的代码，事务管理和 EntityManager 创建、销毁的代码都不再需要开发者关心了。

[**回页首**](http://www.ibm.com/developerworks/cn/opensource/os-cn-spring-jpa/#ibm-pcon)

**更进一步：Spring Data JPA 让一切近乎完美**

通过前面的分析可以看出，Spring 对 JPA 的支持已经非常强大，开发者只需关心核心业务逻辑的实现代码，无需过多关注 EntityManager 的创建、事务处理等 JPA 相关的处理，这基本上也是作为一个开发框架而言所能做到的极限了。然而，Spring 开发小组并没有止步，他们再接再厉，于最近推出了 Spring Data JPA 框架，主要针对的就是 Spring 唯一没有简化到的业务逻辑代码，至此，开发者连仅剩的实现持久层业务逻辑的工作都省了，唯一要做的，就只是声明持久层的接口，其他都交给 Spring Data JPA 来帮你完成！

至此，读者可能会存在一个疑问，框架怎么可能代替开发者实现业务逻辑呢？毕竟，每一个应用的持久层业务甚至领域对象都不尽相同，框架是怎么做到的呢？其实这背后的思想并不复杂，比如，当你看到 UserDao.findUserById() 这样一个方法声明，大致应该能判断出这是根据给定条件的 ID 查询出满足条件的 User 对象。Spring Data JPA 做的便是规范方法的名字，根据符合规范的名字来确定方法需要实现什么样的逻辑。

接下来我们针对前面的例子进行改造，让 Spring Data JPA 来帮助我们完成业务逻辑。在着手写代码之前，开发者需要先 [下载](http://s3.amazonaws.com/dist.springframework.org/release/DATAJPA/spring-data-jpa-1.0.1.RELEASE.zip)Spring Data JPA 的发布包（需要同时下载 Spring Data Commons 和 Spring Data JPA 两个发布包，Commons 是 Spring Data 的公共基础包），并把相关的依赖 JAR 文件加入到 CLASSPATH 中。

首先，让持久层接口 UserDao 继承 Repository 接口。该接口使用了泛型，需要为其提供两个类型：第一个为该接口处理的域对象类型，第二个为该域对象的主键类型。修改后的 UserDao 如下：

**清单 12. Spring Data JPA 风格的持久层接口**

|  |
| --- |
| public interface UserDao extends Repository<AccountInfo, Long> {  public AccountInfo save(AccountInfo accountInfo);  } |

然后删除 UserDaoImpl 类，因为我们前面说过，框架会为我们完成业务逻辑。最后，我们需要在 Spring 配置文件中增加如下配置，以使 Spring 识别出需要为其实现的持久层接口：

**清单 13. 在 Spring 配置文件中启用扫描并自动创建代理的功能**

|  |
| --- |
| <-- 需要在 <beans> 标签中增加对 jpa 命名空间的引用 -->  <jpa:repositories base-package="footmark.springdata.jpa.dao"  entity-manager-factory-ref="entityManagerFactory"  transaction-manager-ref="transactionManager"/> |

至此便大功告成了！执行一下测试代码，然后看一下数据库，新的数据已经如我们预期的添加到表中了。如果要再增加新的持久层业务，比如希望查询出给 ID 的 AccountInfo 对象，该怎么办呢？很简单，在 UserDao 接口中增加一行代码即可：

**清单 14. 修改后的持久层接口，增加一个方法声明**

|  |
| --- |
| public interface UserDao extends Repository<AccountInfo, Long> {  public AccountInfo save(AccountInfo accountInfo);  // 你需要做的，仅仅是新增如下一行方法声明  public AccountInfo findByAccountId(Long accountId);  } |

下面总结一下使用 Spring Data JPA 进行持久层开发大致需要的三个步骤：

1. 声明持久层的接口，该接口继承 Repository，Repository 是一个标记型接口，它不包含任何方法，当然如果有需要，Spring Data 也提供了若干 Repository 子接口，其中定义了一些常用的增删改查，以及分页相关的方法。
2. 在接口中声明需要的业务方法。Spring Data 将根据给定的策略（具体策略稍后讲解）来为其生成实现代码。
3. 在 Spring 配置文件中增加一行声明，让 Spring 为声明的接口创建代理对象。配置了 <jpa:repositories> 后，Spring 初始化容器时将会扫描 base-package 指定的包目录及其子目录，为继承 Repository 或其子接口的接口创建代理对象，并将代理对象注册为 Spring Bean，业务层便可以通过 Spring 自动封装的特性来直接使用该对象。

此外，<jpa:repository> 还提供了一些属性和子标签，便于做更细粒度的控制。可以在 <jpa:repository> 内部使用 <context:include-filter>、<context:exclude-filter> 来过滤掉一些不希望被扫描到的接口。具体的使用方法见 [Spring参考文档](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html#beans-classpath-scanning)。

**应该继承哪个接口？**

前面提到，持久层接口继承 Repository 并不是唯一选择。Repository 接口是 Spring Data 的一个核心接口，它不提供任何方法，开发者需要在自己定义的接口中声明需要的方法。与继承 Repository 等价的一种方式，就是在持久层接口上使用 @RepositoryDefinition 注解，并为其指定 domainClass 和 idClass 属性。如下两种方式是完全等价的：

**清单 15. 两种等价的继承接口方式示例**

|  |
| --- |
| public interface UserDao extends Repository<AccountInfo, Long> { …… }  @RepositoryDefinition(domainClass = AccountInfo.class, idClass = Long.class)  public interface UserDao { …… } |

如果持久层接口较多，且每一个接口都需要声明相似的增删改查方法，直接继承 Repository 就显得有些啰嗦，这时可以继承 CrudRepository，它会自动为域对象创建增删改查方法，供业务层直接使用。开发者只是多写了 "Crud" 四个字母，即刻便为域对象提供了开箱即用的十个增删改查方法。

但是，使用 CrudRepository 也有副作用，它可能暴露了你不希望暴露给业务层的方法。比如某些接口你只希望提供增加的操作而不希望提供删除的方法。针对这种情况，开发者只能退回到 Repository 接口，然后到 CrudRepository 中把希望保留的方法声明复制到自定义的接口中即可。

分页查询和排序是持久层常用的功能，Spring Data 为此提供了 PagingAndSortingRepository 接口，它继承自 CrudRepository 接口，在 CrudRepository 基础上新增了两个与分页有关的方法。但是，我们很少会将自定义的持久层接口直接继承自 PagingAndSortingRepository，而是在继承 Repository 或 CrudRepository 的基础上，在自己声明的方法参数列表最后增加一个 Pageable 或 Sort 类型的参数，用于指定分页或排序信息即可，这比直接使用 PagingAndSortingRepository 提供了更大的灵活性。

JpaRepository 是继承自 PagingAndSortingRepository 的针对 JPA 技术提供的接口，它在父接口的基础上，提供了其他一些方法，比如 flush()，saveAndFlush()，deleteInBatch() 等。如果有这样的需求，则可以继承该接口。

上述四个接口，开发者到底该如何选择？其实依据很简单，根据具体的业务需求，选择其中之一。笔者建议在通常情况下优先选择 Repository 接口。因为 Repository 接口已经能满足日常需求，其他接口能做到的在 Repository 中也能做到，彼此之间并不存在功能强弱的问题。只是 Repository 需要显示声明需要的方法，而其他则可能已经提供了相关的方法，不需要再显式声明，但如果对 Spring Data JPA 不熟悉，别人在检视代码或者接手相关代码时会有疑惑，他们不明白为什么明明在持久层接口中声明了三个方法，而在业务层使用该接口时，却发现有七八个方法可用，从这个角度而言，应该优先考虑使用 Repository 接口。

前面提到，Spring Data JPA 在后台为持久层接口创建代理对象时，会解析方法名字，并实现相应的功能。除了通过方法名字以外，它还可以通过如下两种方式指定查询语句：

1. Spring Data JPA 可以访问 JPA 命名查询语句。开发者只需要在定义命名查询语句时，为其指定一个符合给定格式的名字，Spring Data JPA 便会在创建代理对象时，使用该命名查询语句来实现其功能。
2. 开发者还可以直接在声明的方法上面使用 @Query 注解，并提供一个查询语句作为参数，Spring Data JPA 在创建代理对象时，便以提供的查询语句来实现其功能。

下面我们分别讲述三种创建查询的方式。

**通过解析方法名创建查询**

通过前面的例子，读者基本上对解析方法名创建查询的方式有了一个大致的了解，这也是 Spring Data JPA 吸引开发者的一个很重要的因素。该功能其实并非 Spring Data JPA 首创，而是源自一个开源的 JPA 框架 Hades，该框架的作者 Oliver Gierke 本身又是 Spring Data JPA 项目的 Leader，所以把 Hades 的优势引入到 Spring Data JPA 也就是顺理成章的了。

框架在进行方法名解析时，会先把方法名多余的前缀截取掉，比如 find、findBy、read、readBy、get、getBy，然后对剩下部分进行解析。并且如果方法的最后一个参数是 Sort 或者 Pageable 类型，也会提取相关的信息，以便按规则进行排序或者分页查询。

在创建查询时，我们通过在方法名中使用属性名称来表达，比如 findByUserAddressZip ()。框架在解析该方法时，首先剔除 findBy，然后对剩下的属性进行解析，详细规则如下（此处假设该方法针对的域对象为 AccountInfo 类型）：

* 先判断 userAddressZip （根据 POJO 规范，首字母变为小写，下同）是否为 AccountInfo 的一个属性，如果是，则表示根据该属性进行查询；如果没有该属性，继续第二步；
* 从右往左截取第一个大写字母开头的字符串（此处为 Zip），然后检查剩下的字符串是否为 AccountInfo 的一个属性，如果是，则表示根据该属性进行查询；如果没有该属性，则重复第二步，继续从右往左截取；最后假设 user 为 AccountInfo 的一个属性；
* 接着处理剩下部分（ AddressZip ），先判断 user 所对应的类型是否有 addressZip 属性，如果有，则表示该方法最终是根据 "AccountInfo.user.addressZip" 的取值进行查询；否则继续按照步骤 2 的规则从右往左截取，最终表示根据 "AccountInfo.user.address.zip" 的值进行查询。

可能会存在一种特殊情况，比如 AccountInfo 包含一个 user 的属性，也有一个 userAddress 属性，此时会存在混淆。读者可以明确在属性之间加上 "\_" 以显式表达意图，比如 "findByUser\_AddressZip()" 或者 "findByUserAddress\_Zip()"。

在查询时，通常需要同时根据多个属性进行查询，且查询的条件也格式各样（大于某个值、在某个范围等等），Spring Data JPA 为此提供了一些表达条件查询的关键字，大致如下：

* And --- 等价于 SQL 中的 and 关键字，比如 findByUsernameAndPassword(String user, Striang pwd)；
* Or --- 等价于 SQL 中的 or 关键字，比如 findByUsernameOrAddress(String user, String addr)；
* Between --- 等价于 SQL 中的 between 关键字，比如 findBySalaryBetween(int max, int min)；
* LessThan --- 等价于 SQL 中的 "<"，比如 findBySalaryLessThan(int max)；
* GreaterThan --- 等价于 SQL 中的">"，比如 findBySalaryGreaterThan(int min)；
* IsNull --- 等价于 SQL 中的 "is null"，比如 findByUsernameIsNull()；
* IsNotNull --- 等价于 SQL 中的 "is not null"，比如 findByUsernameIsNotNull()；
* NotNull --- 与 IsNotNull 等价；
* Like --- 等价于 SQL 中的 "like"，比如 findByUsernameLike(String user)；
* NotLike --- 等价于 SQL 中的 "not like"，比如 findByUsernameNotLike(String user)；
* OrderBy --- 等价于 SQL 中的 "order by"，比如 findByUsernameOrderBySalaryAsc(String user)；
* Not --- 等价于 SQL 中的 "！ ="，比如 findByUsernameNot(String user)；
* In --- 等价于 SQL 中的 "in"，比如 findByUsernameIn(Collection<String> userList) ，方法的参数可以是 Collection 类型，也可以是数组或者不定长参数；
* NotIn --- 等价于 SQL 中的 "not in"，比如 findByUsernameNotIn(Collection<String> userList) ，方法的参数可以是 Collection 类型，也可以是数组或者不定长参数；

**使用 @Query 创建查询**

@Query 注解的使用非常简单，只需在声明的方法上面标注该注解，同时提供一个 JP QL 查询语句即可，如下所示：

**清单 16. 使用 @Query 提供自定义查询语句示例**

|  |
| --- |
| public interface UserDao extends Repository<AccountInfo, Long> {  @Query("select a from AccountInfo a where a.accountId = ?1")  public AccountInfo findByAccountId(Long accountId);  @Query("select a from AccountInfo a where a.balance > ?1")  public Page<AccountInfo> findByBalanceGreaterThan(  Integer balance,Pageable pageable);  } |

很多开发者在创建 JP QL 时喜欢使用命名参数来代替位置编号，@Query 也对此提供了支持。JP QL 语句中通过": 变量"的格式来指定参数，同时在方法的参数前面使用 @Param 将方法参数与 JP QL 中的命名参数对应，示例如下：

**清单 17. @Query 支持命名参数示例**

|  |
| --- |
| public interface UserDao extends Repository<AccountInfo, Long> {  public AccountInfo save(AccountInfo accountInfo);  @Query("from AccountInfo a where a.accountId = :id")  public AccountInfo findByAccountId(@Param("id")Long accountId);  @Query("from AccountInfo a where a.balance > :balance")  public Page<AccountInfo> findByBalanceGreaterThan(  @Param("balance")Integer balance,Pageable pageable);  } |

此外，开发者也可以通过使用 @Query 来执行一个更新操作，为此，我们需要在使用 @Query 的同时，用 @Modifying 来将该操作标识为修改查询，这样框架最终会生成一个更新的操作，而非查询。如下所示：

**清单 18. 使用 @Modifying 将查询标识为修改查询**

|  |
| --- |
| @Modifying  @Query("update AccountInfo a set a.salary = ?1 where a.salary < ?2")  public int increaseSalary(int after, int before); |

**通过调用 JPA 命名查询语句创建查询**

命名查询是 JPA 提供的一种将查询语句从方法体中独立出来，以供多个方法共用的功能。Spring Data JPA 对命名查询也提供了很好的支持。用户只需要按照 JPA 规范在 orm.xml 文件或者在代码中使用 @NamedQuery（或 @NamedNativeQuery）定义好查询语句，唯一要做的就是为该语句命名时，需要满足”DomainClass.methodName()”的命名规则。假设定义了如下接口：

**清单 19. 使用 JPA 命名查询时，声明接口及方法时不需要什么特殊处理**

|  |
| --- |
| public interface UserDao extends Repository<AccountInfo, Long> {  ......    public List<AccountInfo> findTop5();  } |

如果希望为 findTop5() 创建命名查询，并与之关联，我们只需要在适当的位置定义命名查询语句，并将其命名为 "AccountInfo.findTop5"，框架在创建代理类的过程中，解析到该方法时，优先查找名为 "AccountInfo.findTop5" 的命名查询定义，如果没有找到，则尝试解析方法名，根据方法名字创建查询。

**创建查询的顺序**

Spring Data JPA 在为接口创建代理对象时，如果发现同时存在多种上述情况可用，它该优先采用哪种策略呢？为此，<jpa:repositories> 提供了 query-lookup-strategy 属性，用以指定查找的顺序。它有如下三个取值：

* create --- 通过解析方法名字来创建查询。即使有符合的命名查询，或者方法通过 @Query 指定的查询语句，都将会被忽略。
* create-if-not-found --- 如果方法通过 @Query 指定了查询语句，则使用该语句实现查询；如果没有，则查找是否定义了符合条件的命名查询，如果找到，则使用该命名查询；如果两者都没有找到，则通过解析方法名字来创建查询。这是 query-lookup-strategy 属性的默认值。
* use-declared-query --- 如果方法通过 @Query 指定了查询语句，则使用该语句实现查询；如果没有，则查找是否定义了符合条件的命名查询，如果找到，则使用该命名查询；如果两者都没有找到，则抛出异常。

**Spring Data JPA 对事务的支持**

默认情况下，Spring Data JPA 实现的方法都是使用事务的。针对查询类型的方法，其等价于 @Transactional(readOnly=true)；增删改类型的方法，等价于 @Transactional。可以看出，除了将查询的方法设为只读事务外，其他事务属性均采用默认值。

如果用户觉得有必要，可以在接口方法上使用 @Transactional 显式指定事务属性，该值覆盖 Spring Data JPA 提供的默认值。同时，开发者也可以在业务层方法上使用 @Transactional 指定事务属性，这主要针对一个业务层方法多次调用持久层方法的情况。持久层的事务会根据设置的事务传播行为来决定是挂起业务层事务还是加入业务层的事务。具体 @Transactional 的使用，请参考[Spring的参考文档](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/transaction.html#transaction-declarative-annotations)。

**为接口中的部分方法提供自定义实现**

有些时候，开发者可能需要在某些方法中做一些特殊的处理，此时自动生成的代理对象不能完全满足要求。为了享受 Spring Data JPA 带给我们的便利，同时又能够为部分方法提供自定义实现，我们可以采用如下的方法：

* 将需要开发者手动实现的方法从持久层接口（假设为 AccountDao ）中抽取出来，独立成一个新的接口（假设为 AccountDaoPlus ），并让 AccountDao 继承 AccountDaoPlus；
* 为 AccountDaoPlus 提供自定义实现（假设为 AccountDaoPlusImpl ）；
* 将 AccountDaoPlusImpl 配置为 Spring Bean；
* 在 <jpa:repositories> 中按清单 19 的方式进行配置。

**清单 20. 指定自定义实现类**

|  |
| --- |
| <jpa:repositories base-package="footmark.springdata.jpa.dao">  <jpa:repository id="accountDao" repository-impl-ref=" accountDaoPlus " />  </jpa:repositories>  <bean id="accountDaoPlus" class="......."/> |

此外，<jpa:repositories > 提供了一个 repository-impl-postfix 属性，用以指定实现类的后缀。假设做了如下配置：

**清单 21. 设置自动查找时默认的自定义实现类命名规则**

|  |
| --- |
| <jpa:repositories base-package="footmark.springdata.jpa.dao"  repository-impl-postfix="Impl"/> |

则在框架扫描到 AccountDao 接口时，它将尝试在相同的包目录下查找 AccountDaoImpl.java，如果找到，便将其中的实现方法作为最终生成的代理类中相应方法的实现。

[**回页首**](http://www.ibm.com/developerworks/cn/opensource/os-cn-spring-jpa/#ibm-pcon)

**结束语**

本文主要介绍了 Spring Data JPA 的使用，以及它与 Spring 框架的无缝集成。Spring Data JPA 其实并不依赖于 Spring 框架，有兴趣的读者可以参考本文最后的"参考资源"进一步学习。